(2)

Compiler-Assisted Full Checkpointing

Chung-Chi Jim Li, Elliot M. Stewart, and *W. Kent Fuchs*

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 West Main Street
Urbana, IL 61801-2307

Correspondent: W. Kent Fuchs
Tel: (217)333-9371
FAX: (217)244-5686
Email: fuchs@crhc.uiuc.edu

Summary

This paper describes a compiler-based approach to efficient checkpointing for process recovery. Our implementation is transparent to both the programmer and the hardware. The compiler-generated *sparse potential checkpoint* code maintains the desired checkpoint interval. *Adaptive checkpointing* reduces the size of the checkpoints. *Training* is used to select low-cost, high-coverage potential checkpoints. The problem of selecting potential checkpoints is shown to be NP-complete and a heuristic algorithm is introduced that determines a quick suboptimal solution. These compiler-assisted checkpointing techniques have been implemented in a modified version of the GNU C (GCC) compiler. Experiments involving the CATCH GCC compiler on a Sun SPARC workstation are summarized.

Index Terms: compiler-assisted checkpointing, fault recovery, fault-tolerant computing

This research was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy, and U.S. Air Force) under Grant N00014-90-J-1270, and in part by the Department of the Navy managed by the Office of Naval Research under Grant N00014-91-J-1283. A brief initial version of this paper was presented at the Twentieth International Symposium on Fault-Tolerant Comouting, 1990.

1. Introduction

1.1. Background

Checkpointing and rollback recovery is a standard technique used in providing a fault-tolerant computing environment. With checkpointing, if a fault is detected by the computer system, the application is then rolled back to the most recent checkpoint where the state of the system is known to be correct. Checkpointing schemes can be broadly divided into two categories, depending on how the checkpoint information is maintained. One scheme, called *full checkpointing*, involves saving the entire state space of the application for every checkpoint. The other scheme, called *incremental checkpointing*, creates major checkpoints which save the entire state space and minor checkpoints which save the difference between the current and previous state space. This paper examines the application of compiler-based techniques for implementing full checkpointing.

Researchers have made use of compilers to help facilitate fault-tolerant schemes in a variety of ways. Alewine *et al.* [1] implemented a compiler-assisted multiple-instruction retry approach which could tolerate a wide class of code execution failures. The compiler generates code in such a way as to eliminate possible on-path and branch hazards that might result when instruction retry is performed. Long, Fuchs, and Abraham [2] created a compiler-assisted approach for static checkpoint insertion which maintains the desired checkpoint interval and reproducible checkpoint locations. Balasubramanian and Banerjee [3] utilize the compiler to aid in synthesizing algorithm-based checking techniques for general applications. Wilken and Kong [4] developed a method of concurrent checking of memory accesses by means of a signature that is embedded into different types of data structures.

Research pertaining to classical checkpointing and rollback recovery is heavily documented. However, there is little published work on utilizing compilers in generating checkpoints. Related to this area is the work done by Chandy and Ramamoorthy [5]. They formulated a graph-theoretic method which guides a programmer in the decision of where to insert checkpoints in a program. The program is broken down into distinct tasks. Based on factors such as execution time, checkpoint time, and failure rates, their algorithm can determine optimal locations, between tasks, for checkpoint placement. By identifying the optimal checkpoint locations, the maximum checkpoint time, the expected checkpoint time, or the expected run time can be minimized. A similar approach was pursued by Toueg and Babaoğlu [6], and Upadhyaya and Saluja [7]. The problem with this approach, as pointed out by Chandy and Ramamoorthy [5], is that the cost of analysis incurred by the programmer can be excessive.

1.2. The CATCH GCC Compiler

This paper details three different compiler-assisted techniques for implementing full checkpointing. The checkpointing techniques are called *compiler-assisted techniques for checkpointing (CATCH)*. The checkpointing scheme is transparent to the programmer and requires no modifications to the computer architecture. The GNU C compiler (GCC) version 2.4.3 [8] was modified in order to implement the checkpointing schemes. Figure 1 shows the modifications made to the compiler. Highlighted areas represent additions to the original compiler. The purpose of the CATCH filter is to insert code into the user program with supporting functions defined in the CATCH libraries. The experiments detailed in this paper were run on a Sun SPARC workstation (sparc2, UNIX SunOS Release 4.1.3).

2. Checkpoint Interval Maintenance

This section describes how the compiler inserts *sparse potential checkpoints* in the program to maintain the desired checkpoint interval. An optimal checkpoint interval can be determined by following one of the

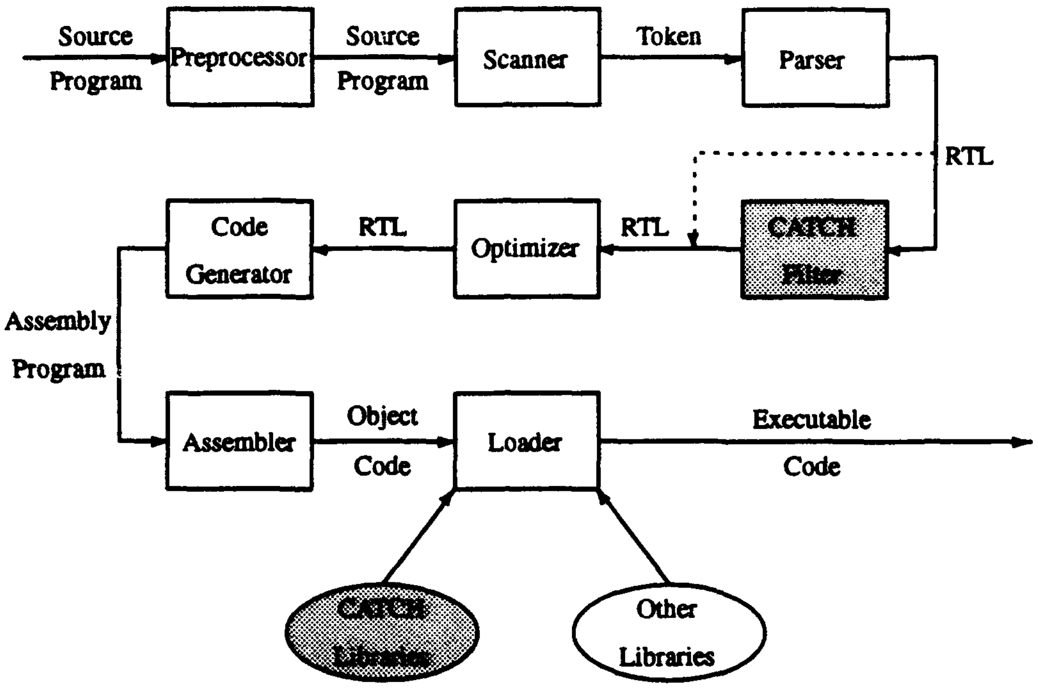

Figure 1: Stages of the CATCH compiler

documented standard analysis methods [9-17].

2.1. Potential Checkpoints

The fundamental aspect of our approach is to have the compiler insert code into the user program so that the resulting executable can periodically establish checkpoints. Polling is used to determine elapsed time, since not all general purpose multi-user systems have user accessible interrupt timers. Polling also keeps the implementation easier to port to other computer systems. The code that establishes checkpoints is a subroutine named *_checkpoint*(). It is comparable to Taylor and Wright's [18] *_Establish*() subroutine and the *rfork*() subroutine described by Smith and Ioannidis [19]. One notable difference from the *rfork*() approach is that *_checkpoint*() does not require *fork*() to create a child process before establishing a checkpoint.

An inherent difficulty of this approach is finding the proper places in the code to take checkpoints, so as to maintain a regular checkpoint interval. For example, in the following small block of code the compiler may not be able to ascertain where to insert the *_checkpoint*() subroutine due to the fact that the loop upper bound is unknown at compile time.

```
scanf("%d", &bound);
sum = 0;
for (i = 0; i < bound; i++) {
     sum += f(i);
}
printf("%d\n", sum);
```

If the compiler inserts the *_checkpoint*() subroutine inside the loop, then the time required to establish a checkpoint might be prohibitive. On the other hand, if the compiler places the *_checkpoint*() call after the loop, the desired checkpoint interval may be exceeded if the loop code executes for a long period of time.

The CATCH GCC compiler resolves this dilemma by locating places in the program where the clock should be examined in order to decide whether a checkpoint needs to be taken or not. These locations are called *potential checkpoints*. In our implementation, potential checkpoints are at the beginning of subroutines and at the first line inside a loop. The code that handles the potential checkpoint functionality is a subroutine called *_potential*(). The *_potential*() subroutine acts somewhat like a software monitor [20].

The *_potential*() subroutine performs two functions:

1. Maintaining checkpoint intervals: keep the time elapsed between established checkpoints as close to the optimal checkpoint interval (denoted T) as possible.
2. Minimization of resources: minimize the time and/or space required of the checkpointing activities.

Most system level checkpointing implementations address only the first point and maintain a consistent, rigid checkpoint interval. Yet, as will be shown in Section 3 and Section 4, by relaxing the constraint of a rigid checkpoint interval, a significant saving of system resources is sometimes possible.

Every encounter of a potential checkpoint allows the user program to take a checkpoint. If the time since the last established checkpoint exceeds T, then a checkpoint is established at that encountered potential checkpoint. Since potential checkpoint locations are frequently encountered (on the order of hundreds of microseconds) and T is usually on the order of minutes, the checkpoint interval is consistently maintained.

2.2. Sparse Potential Checkpoints

Even though by placing *_potential*() at the beginning of subroutines and within loops helps to maintain a rigid checkpoint interval, nonetheless there is a high performance overhead in the frequent polling that results. To reduce the excessive polling, and hence reduce the performance overhead, a technique called *sparse potential checkpoint* is used. The following two line code segment replaces the original call to *_potential*():

```
if (--counter <= 0)
    _potential();
```

where *counter* is initially set to a *reduction factor l*. As a result, only one call to *_potential*(), per l potential checkpoint encounters, is generated. The average checkpoint interval is still maintained close to T, since the time between sparse potential checkpoints, even with l on the order of thousands, is relatively short.

Table I outlines the results of an experiment comparing the cost of the original and sparse potential checkpoint code on a Sun SPARC. The first column is the execution time of the UNIX *time*() library subroutine, which consumes most of the time in *_potential*() (second column). The third column represents the time required to execute the sparse potential checkpoint code if the counter has not yet reached zero. Otherwise, the sparse potential checkpoint code takes 37.19 μs to execute. The average cost of the sparse potential checkpoint code can be calculated from the equation $0.677 + \frac{37.19}{l}$. As an example, when l is set to 1000, the cost of the sparse potential checkpoint code (fourth column) is significantly less than the original potential checkpoint code (second column). The lower cost is prominent in reducing the polling time overhead.

Table I: Cost comparison of the original and sparse potential checkpoint code (μs)

time()	**_potential()**	**$l = \infty$**	**$l = 1000$**
36.51	37.19	0.677	0.704

2.3. Experimental Results

Two factors account for the run-time overhead of a program compiled by the CATCH GCC compiler. The first factor is the time required to perform the actual checkpoint saves. The second factor is attributed to the polling time, called the potential checkpoint time. The storage device used in the experiments was an IPI disk on a Sun file server with a 6 Mbytes/sec access time. In all, ten benchmark programs were compiled with the CATCH GCC compiler and subsequently run. Specifically, the following are the benchmarks used, some of which were first converted from FORTRAN to C with the f2c translator developed by AT&T Bell Laboratories and Bellcore.

RKF: uses the Runge-Kutta-Fehlberg method [21] in solving the ordinary differential equation $y' = x + y$, $y(0) = 2$. The computations performed are floating point intensive.

CONVLV: finds the convolution of 1024 signals with one response using the FFT algorithm [22]. Each signal length was 256 bytes while the length of the response was 99 bytes. The entire data set was greater than 1 Mbytes, however, the memory-resident data set occupied only a few Kbytes.

LUDCMP: is an LU decomposition algorithm [21] that decomposes 100 randomly generated matrices with size uniformly distributed between 1 and 100. A characteristic of the benchmark is that memory blocks are allocated before a new matrix is read in and then are subsequently deallocated after the result is written out.

ESPRESSO: is a SPEC benchmark program [23] developed at the University of California at Berkeley that performs boolean function minimization. It is an integer benchmark with many short loops and recursive functions.

MDLJSP2: obtains the solution to the equation of motion for a model of 500 atoms interacting through the idealized Lenard-Jones potential. The program is from the collection of SPEC benchmark programs.

FPPPP: is a quantum chemistry SPEC benchmark which heavily involves double-precision floating point operations. The code is characterized by its very large basic blocks.

ORA: is a CPU intensive floating point SPEC benchmark. ORA traces rays through an optical system comprised of different surfaces. Only a small amount of I/O is performed.

MATRIX300: is one of the SPEC benchmarks that performs a variety of matrix multiplications, including matrix transposes using Linpack routines SGEMV, SGEMM, and SAXPY, on matrices on the order of 300.

TOMCATV: is a mesh generation program and is also one of the SPEC benchmarks. The program involves very little I/O.

NASA7: is one of the SPEC benchmarks and is comprised of seven kernels. Each kernel generates its own input data, performs the kernel, and compares the result for correctness. The kernels involve a variety of computations dealing with arrays and matrices.

Table II shows the results of the first experimental run involving two versions of each program. The "original" version represents the program compiled by the unmodified GCC compiler. The "normal" version represents the program compiled by the CATCH GCC compiler. The parameters for the "normal" version are as follows. Each benchmark has $l = 1000$. For the shorter benchmarks like RKF, CONVLV, LUDCMP, and ESPRESSO, the value of T was set to 10 seconds. MDLJSP2 had $T = 120$ seconds, both FPPPP and ORA had $T = 150$ seconds, MATRIX300 had $T = 180$ seconds, TOMCATV had $T = 90$ seconds, and lastly NASA7 had $T = 900$ seconds. To determine how much time can be attributed to polling in the "normal" version, another version of the program (not shown in the table) was compiled with $T = \infty$ (actually, a very large number). The consequence is that this version will not establish any checkpoints, hence the longer run time can be solely accounted for by the potential checkpoints.

The checkpoint time is expressed as the product of two numbers. The first number is the average time used in establishing a single checkpoint while the second number represents the total number of checkpoints taken during one run averaged over multiple test runs. The total time overhead merit is expressed as a percentage of the execution time of the "original" version. For most of the benchmarks, the overhead is near or below the 10% mark. The time overhead for some of the benchmarks, though, is not satisfactory and the methods described in the following sections must be employed to make the overhead tolerable.

Table II: Performance of the "original" and "normal" versions

program	**version**	**total execution time (secs)**	**average checkpoint size (bytes)**	**potential checkpoint time (secs)**	**checkpoint time (secs*#)**	**total time overhead (%)**
RKF	original	21.0	-	-	-	-
	normal	22.8	19424.0	1.00	0.40*2	8.57
CONVLV	original	17.8	-	-	-	-
	normal	20.6	49729.6	2.20	0.30*2	15.73
LUDCMP	original	24.0	-	-	-	-
	normal	32.0	108840.3	7.00	0.33*3	33.33
ESPRESSO	original	27.0	-	-	-	-
	normal	48.2	331955.2	19.20	0.48*4.2	78.52
MDLJSP2	original	530.8	-	-	-	-
	normal	609.4	211653.9	77.80	0.16*5	14.81
FPPPP	original	645.6	-	-	-	-
	normal	658.4	359358.4	12.00	0.20*4	1.98
ORA	original	727.0	-	-	-	-
	normal	733.6	55068.0	5.20	0.35*4	0.91
MATRIX300	original	1053.0	-	-	-	-
	normal	1173.4	2189567.7	113.40	1.17*6	11.43
TOMCATV	original	377.6	-	-	-	-
	normal	405.2	3730204.0	16.00	2.90*4	7.31
NASA7	original	3662.0	-	-	-	-
	normal	3879.0	2924748.0	212.80	1.05*4	5.93

3. Checkpoint Size Reduction

3.1. Data Compression

One method of reducing checkpoint size is to perform data compression on the checkpoint files before they are written out. We implemented an LZW data compression [24], which provided significant checkpoint size reduction that averaged 48.9% over all checkpoints for all the benchmark programs. However, the compression algorithm is computationally intensive. As a result, we developed alternative run-time methods for enhancing checkpoint performance.

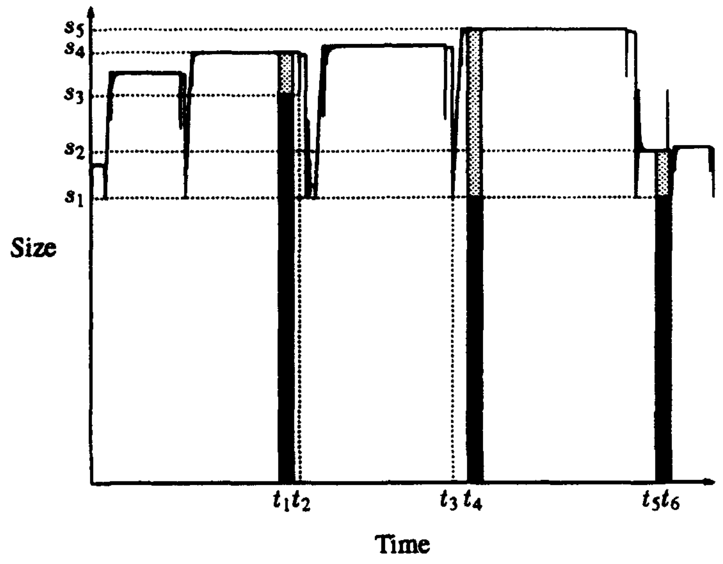

Figure 2: Variation in checkpoint size for the "small" version of LUDCMP

3.2. Adaptive Checkpointing

The motivation for an adaptive checkpointing scheme, which allows small deviations in T, is illustrated in Figure 2. Displayed is the checkpoint size trace, over time, of a "small" version of LUDCMP, which exhibits large variations in memory usage. As an example, suppose three checkpoints are taken at time t_1, t_4, and t_5 with corresponding sizes s_4, s_5, and s_2. If the first checkpoint could be delayed to t_2, the second shifted back to t_3, and the third delayed until t_6, then the resulting checkpoints would be reduced to size s_3, s_1, and s_1 respectively. *Adaptive checkpointing* is the method of allowing a window of opportunity to take a checkpoint in order to take advantage of variations in memory use.

The adaptive checkpointing scheme uses a decision algorithm embedded in $\_potential()$. The algorithm can be viewed pictorially as a *transition graph*, as seen in Figure 3. The transition graph is constructed dynamically during program execution. When a potential checkpoint is first encountered, a new node in the graph is created; otherwise the corresponding node is located. Associated with each node is a *cost* (the

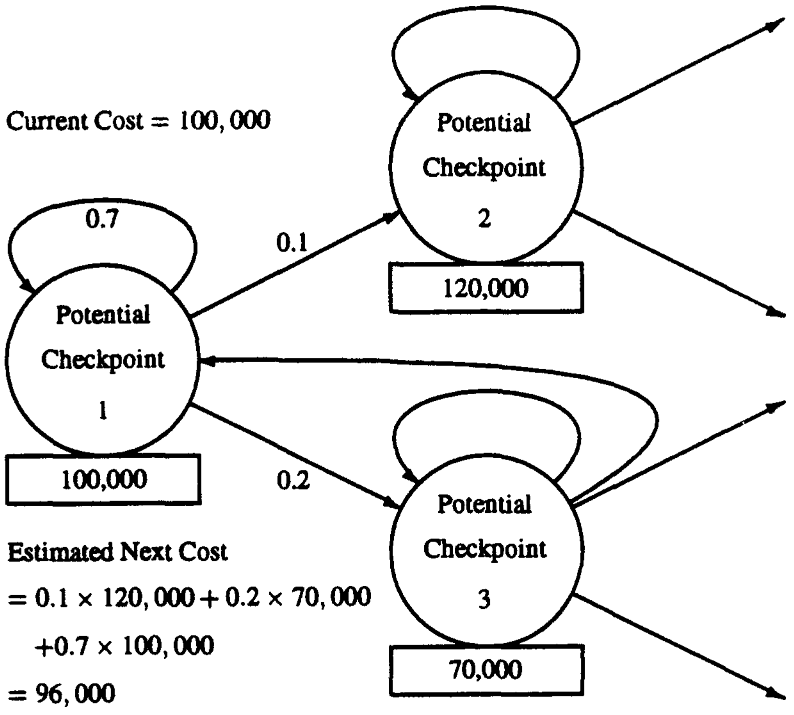

Figure 3: Adaptive checkpointing transition graph

number in the box below each node in Figure 3), which in our implementation is the average size of the checkpoint that would be established at that particular potential checkpoint. Each edge has a *transition probability*, which represents the probability of encountering next the potential checkpoint attached to that edge. Both the cost and transition probability are updated during a potential checkpoint encounter.

There are two factors that guide the decision of when to take a checkpoint: time (T, w, and the current time) and cost (the current cost and the *estimated ne⁻ᵗ cost*). The variable w is the *checkpointing window size*, which delimits the range of time where a checkpoint can be taken. The estimated next cost is simply the weighted average of the costs of all the immediate successor nodes of the current node. The guideline for establishing a checkpoint is stated as:

```
if (the time since the last established checkpoint >= T)
    _checkpoint();
else if ((T - w <= time since the last established checkpoint)
    and (the estimated next cost > the current cost))
    _checkpoint();
```

```
else
    wait until the next encounter of a potential checkpoint
    to see if a checkpoint should be established;
```

To illustrate the functionality of the transition graph in Figure 3, consider an example where the current potential checkpoint is node 1 and the current cost is 100,000. If the elapsed time since the last checkpoint falls in the window (i.e., between $T - w$ and T), the decision to take a checkpoint will be postponed because the estimated next cost (96,000) is lower than the current cost. If, on the other hand, the elapsed time was equal to or greater than T, then the algorithm would immediately force a checkpoint to be taken.

3.3. Experimental Results

Table III summarizes the results of the "adaptive" version. For the experiments, $w = 1$ second for RKF, CONVLV, LUDCMP, and ESPRESSO and $w = 6$ seconds for the remaining benchmarks. In our experiments, values of w less than 10% of T maintained a consistent checkpoint interval. The "adaptive" version produced smaller checkpoints for only LUDCMP. The other benchmarks are written in such a way that memory use is relatively constant. As a result, the "adaptive" version will not encounter locations where memory use is markedly lower and hence be able to establish a smaller checkpoint. The increase in checkpoint size is due to the transition graph information that must also be saved.

4. Potential Checkpoint Time Reduction

This section outlines the *training* method that reduces the polling time of the potential checkpoints. The training method is inspired by Chang and Hwu's profiling technique [26]. A new subroutine called *_snapshot*(), which maintains statistical information on the checkpoint time and potential checkpoint time, replaces the sparse potential checkpoint code. The execution time overhead for the *_snapshot*() subroutine

Table III: Performance of the "adaptive" version

program	total execution time (secs)	average checkpoint size (bytes)	potential checkpoint time (secs)	checkpoint time (secs*#)	total time overhead (%)
RKF	22.8	19468.0	1.00	0.40*2	8.57
CONVLV	19.6	50052.8	1.20	0.30*2	10.11
LUDCMP	31.8	64551.2	7.20	0.20*3	32.50
ESPRESSO	60.0	332252.8	30.12	0.48*6	122.22
MDLJSP2	631.8	213554.7	100.00	0.20*5	19.03
FPPPP	676.6	361252.8	30.00	0.25*4	4.80
ORA	740.2	55324.0	11.80	0.35*4	1.82
MATRIX300	1188.0	2189956.0	127.75	1.17*6.2	12.82
TOMCATV	405.6	3730716.0	16.40	2.90*4	7.42
NASA7	3916.4	2928744.0	250.20	1.05*4	6.95

is 30.87 μs per encounter. The training technique is comprised of three steps:

1. Compile a "training" version.
2. Run the "training" version to acquire a profile of the checkpoint size over time.
3. Compile the "trained" version of the program, utilizing the information gathered in step 2.

4.1. Sampling

The purpose of the *_snapshot*() subroutine is to maintain for every potential checkpoint two statistics. It records the number of times a potential checkpoint is encountered in addition to the average checkpoint size, as if checkpoints were established at this potential checkpoint location. The information is written out to an accounting file after every l encounters of potential checkpoints. Also, the "training" version records in the accounting file the total execution time of the program.

The *sampling* phase (step 2) is pictorially represented in Figure 4, which shows a trace of the checkpoint size for the "small" version of the LUDCMP benchmark. The trace is subdivided into ten time intervals labeled a_1 through a_{10}, each containing 18300 encounters of potential checkpoints (i.e., $l = 18300$). The labels S_4 and S_{14} represent the fourth and fourteenth potential checkpoints (the remaining labels are omitted

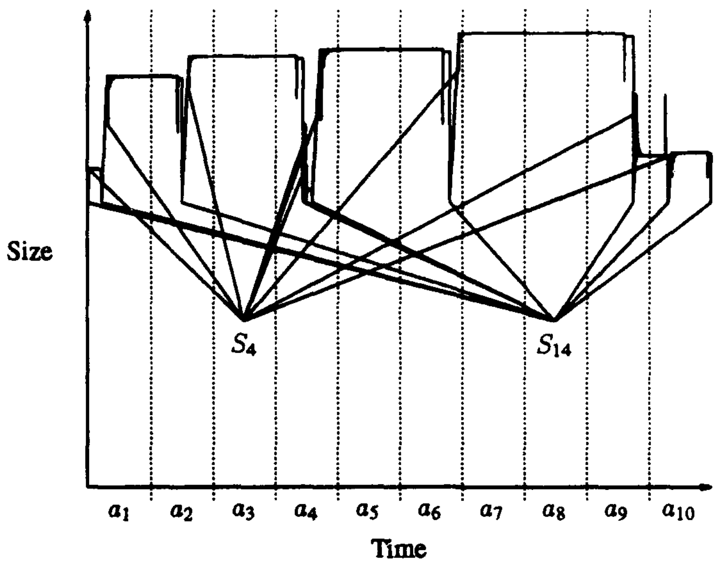

Figure 4: Sampling the checkpoint size by potential checkpoints

for clarity). The lines emitting from S_4 and S_{14} show the locations where the two respective potential checkpoints are encountered. As an example, if a checkpoint should be established at S_4 or S_{14}, it would be advantageous to select S_{14} since the average checkpoint size is less than that of S_4.

4.2. Analysis

The final phase of the training method is the *analysis* phase (step 3), where the compiler processes the accounting information provided by the sampling phase. There are two functions to be performed in this phase: 1) assign a cost to every potential checkpoint, and 2) determine a minimum cost subset of all potential checkpoints that can fully cover the time intervals. The process of finding a minimum cost subset is aided by the use of a bipartite graph as illustrated in Figure 5. The nodes on the left-hand side represent the potential checkpoints S_i, $1 \leq i \leq m$. The nodes on the right-hand side represent the time intervals a_j, $1 \leq j \leq n$. A cost pair (P_i, C_i) is calculated and assigned to each S_i, where P_i is the polling cost and C_i is the cost of establishing a checkpoint for potential checkpoint i. A line joins an S_i to an a_j if that S_i is encountered

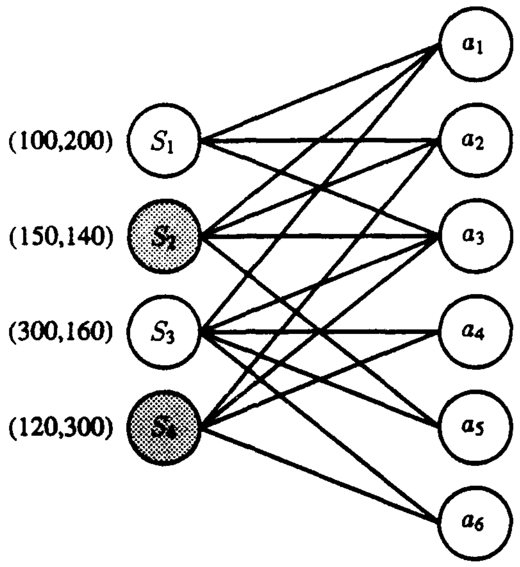

Figure 5: Weighted Minimum Cover (WMC) problem

during that time interval. The cost of a subset $X = \{S_{i_k} | 1 \leq k \leq m'\}$ is defined as:

$$Cost(X) = \sum_{k=1}^{m'} P_{i_k} + \frac{\sum_{k=1}^{m'} P_{i_k} C_{i_k}}{\sum_{k=1}^{m'} P_{i_k}}$$

where $i_k \in \{1, \ldots, m\}$. The first term of the cost function is the total polling time, contributed by the individual potential checkpoints. The second term is based on the premise that the more frequently a potential checkpoint is encountered, the greater the possibility that a checkpoint will be established there. Hence, the total checkpoint time (second term) is a weighted average of the checkpoint time of each potential checkpoint.

For a particular checkpoint i, the value of P_i can be calculated as the product of the time overhead of the sparse potential checkpoint code and the number of encounters. The value of C_i is calculated as the product of the checkpoint size and the number of checkpoints established during program execution. The number of checkpoints established is determined by dividing the net execution time (total execution time minus the time spend in *_snapshot*()) by T. The time overhead value of the sparse potential checkpoint code needed in the P_i calculation is obtained from Table I.

4.3. Weighted Minimum Cover (WMC) Problem

After the costs are calculated, the final task of finding a minimum cost subset of potential checkpoints that covers all the time intervals can be addressed. This task is called the *Weighted Minimum Cover* (WMC) problem. As an example, the minimum cost subset in Figure 5 is comprised of the two shaded potential checkpoints.

Before the WMC problem is formally stated, certain variables need to be defined. Consider each S_i a subset of $A = \{a_j | 1 \leq j \leq n\}$. Let $S \triangleq \{(S_i, P_i, C_i) | S_i \subseteq A, P_i$ and C_i are positive real numbers, $1 \leq i \leq m\}$, f a polynomial time cost function with bounded value ranging from 2^S to non-negative real numbers, and K is a positive real number. A set $X \subseteq S$ is considered a *cover* of A if $\{\cup S_i = A|$ for all S_i in $X\}$.

The WMC decision problem can be formally stated as follows: Given a 4-tuple (A, S, f, K), does there exist a subset $X \subseteq S$ such that X is a cover of A and $f(X) \leq K$? There might be more than one unique subset that satisfies the previously stated problem constraints. With this fact in mind, a corresponding WMC optimization problem can be stated as follows: Given a 3-tuple (A, S, f), find a subset $X \subseteq S$ such that X is a cover of A, and $f(X) \leq f(Y)$ for every Y that is a cover of A. By proving that the WMC decision problem is NP-complete, it is straightforward to show that the WMC optimization problem is NP-hard.

Property: The WMC decision problem is NP-complete.

Proof: To prove that the WMC decision problem is NP-complete, it is adequate enough to show that it is in NP-space and is NP-hard. For a given X, the process of verifying X is a cover of A, calculating $f(X)$, and subsequently comparing the result to K can all be accomplished in polynomial time. Thus, the WMC decision problem is in NP. If the function f is confined to positive solutions, i.e. to $f(X) = |X|$, then the WMC decision problem becomes the Minimum Cover decision problem which is known to be NP-complete [27]. Therefore, the WMC decision problem is also NP-hard.

4.4. A Heuristic Algorithm

The number of potential checkpoints in a program can range from tens to hundreds and the number of time intervals is usually of the order of thousands. As a result, the process of finding an optimal solution to the WMC problem can be computationally expensive. However, obtaining an optimal solution is neither critical nor necessary because the costs are based on estimates, and secondly, in our experiments suboptimal solutions perform well.

The heuristic algorithm detailed in this section is designed with two factors in mind: 1) some potential checkpoints are infrequently encountered (e.g., those at the beginning, the end, and in blocks of code dealing with error cases), and 2) by not covering all the time intervals, performance may be only slightly impacted. The algorithm processes and selects potential checkpoints based on a *coverage factor* lower bound (LOWMARK) and upper bound (HIGHMARK). The coverage factor of a subset $X \subseteq S$ is expressed as the number of time intervals covered by the union of the first components of its elements divided by the total number of time intervals. For example, in Figure 5, the subset $X = \{S_2, S_4\}$ has a coverage factor of 1.0 since all the time intervals (a_1 through a_6) are covered. The experiments performed used a value of 0.5 for LOWMARK and 0.9 for HIGHMARK. The algorithm is described as follows:

```
delete all potential checkpoints with coverage factor
    less than LOWMARK;
use a sorting algorithm to order potential checkpoints by
    the cost Pi+Ci;
X = next candidate solution;
while(X is not empty) {
    if (coverage factor of X >= HIGHMARK)
        break;
    else
        X = next candidate solution;
}
if (X is empty)
    X = S;
record X permanently;
```

The first part of the algorithm eliminates all the potential checkpoints that are infrequently encountered. The remaining potential checkpoints are considered high coverage (greater than LOWMARK). Hence, the task of finding a subset of potential checkpoints with a combined coverage factor greater than HIGHMARK will be simplified, though, the algorithm could still take exponential time in the worst case. The candidate solution may be generated in either a depth-first or breadth-first manner.

4.5. Experimental Results

There are two aspects to consider when evaluating the performance of the training method. The first aspect is the amount of system resources consumed by the compilation and execution of the "training" version. The second aspect to evaluate is the performance of the actual "trained" version. Table IV summarizes the results of the first aspect. The training time refers to the total execution time of the "training" version and the analysis time is the time to run the heuristic algorithm, including the I/O time to read the sampling data from the accounting file. The cover set shows the number of potential checkpoints retained by the heuristic algorithm. The coverage provided by the cover set is represented as a percentage. For large and/or long running programs (ESPRESSO, MDLJSP2, MATRIX300, TOMCATV, NASA7) the training time is limited by a compiler option. The reason for the time restriction is because the accounting file can grow dramatically large in addition to the training time becoming excessively long.

The results of the "trained" version, along with the other versions, are shown in Table V. The input data used for the "trained" version experiments differed from the input data used during the training step. For all the benchmarks, except MDLJSP2, the polling time attributed to the potential checkpoints is significantly

Table IV: Performance of the "training" version

program	# of potential checkpoints	# of selected intervals	training time (secs)	file size (bytes)	analysis time (msecs)	cover set	coverage (%)
RKF	5	2,577	93	113400	233	{1}	100
CONVLV	20	4,317	137	525112	1008	{6}	94.8
LUDCMP	18	10,502	315	979092	1916	{7,9}	93.0
ESPRESSO	1055	1,365	60	9614400	16269	{50,810,918}	100
MDLJSP2	83	7,342	360	2639028	4875	{81}	100
FPPPP	75	2,825	336	1633720	2983	{4}	97.7
ORA	9	3,648	957	277260	525	{6}	99.1
MATRIX300	24	5,336	180	946812	1691	{20}	98.2
TOMCATV	19	2,057	90	227384	10349	{6,8,12,15,17}	95.3
NASA7	149	19,437	900	2705032	4883	{15}	97.0

reduced and in some cases completely negligible. The reason MDLJSP2 did not perform better is that the number of potential checkpoints was only reduced from 83 to 81. The principal feature of MDLJSP2 that caused the "trained" version to perform badly is the lack of a small set of dominant potential checkpoints. Most of the potential checkpoints were retained to help cover the time intervals.

A correlation exists between certain code characteristics and the performance of the CATCH compiler in training mode. Code with a significant amount of recursion usually results in little performance improvement. The potential checkpoint at the beginning of the recursive subroutine can be encountered quite often, thus heavily increasing the polling time. Conversely, code comprised of loops with many iterations and large loop body size can perform well with the CATCH compiler. In theses cases, the potential checkpoint execution time will not dominate the execution time of the loop iteration. Even with nested loops the performance can be satisfactory, since the training technique can usually remove the majority of the potential checkpoints located at different nested levels to reduce the polling. Benchmarks that possessed these favorable code characteristics and performed well under training were FPPPP, ORA, MATRIX300, TOMCATV, and NASA7.

Table V: Performance for all versions

program	version	total execution time (secs)	average checkpoint size (bytes)	potential checkpoint time (secs)	checkpoint time (secs*#)	total time overhead (%)
RKF	original	21.0	-	-	-	-
	normal	22.8	19424.0	1.00	0.40*2	8.57
	adaptive	22.8	19468.0	1.00	0.40*2	8.57
	trained	22.0	19340.0	0.00	0.50*2	4.76
CONVLV	original	17.8	-	-	-	-
	normal	20.6	49729.6	2.20	0.30*2	15.73
	adaptive	19.6	50052.8	1.20	0.30*2	10.11
	trained	19.2	49780.0	0.80	0.30*2	7.87
LUDCMP	original	24.0	-	-	-	-
	normal	32.0	108840.3	7.00	0.33*3	33.33
	adaptive	31.8	64551.2	7.20	0.20*3	32.50
	trained	27.4	131320.8	3.00	0.20*2	14.17
ESPRESSO	original	27.0	-	-	-	-
	normal	48.2	331955.2	19.20	0.48*4.2	78.52
	adaptive	60.0	332252.8	30.12	0.48*6	122.22
	trained	32.6	348672.0	4.40	0.60*2	20.74
MDLJSP2	original	530.8	-	-	-	-
	normal	609.4	211653.9	77.80	0.16*5	14.81
	adaptive	631.8	213554.7	100.00	0.20*5	19.03
	trained	612.4	211652.6	80.80	0.16*5	15.37
FPPPP	original	645.6	-	-	-	-
	normal	658.4	359358.4	12.00	0.20*4	1.98
	adaptive	676.6	361252.8	30.00	0.25*4	4.80
	trained	647.2	359492.0	1.00	0.20*3	0.25
ORA	original	727.0	-	-	-	-
	normal	733.6	55068.0	5.20	0.35*4	0.91
	adaptive	740.2	55324.0	11.80	0.35*4	1.82
	trained	728.8	54844.0	0.40	0.35*4	0.25
MATRIX300	original	1053.0	-	-	-	-
	normal	1173.4	2189567.7	113.40	1.17*6	11.43
	adaptive	1188.0	2189956.0	127.75	1.17*6.2	12.82
	trained	1058.8	2189572.0	0.00	1.16*5	0.55
TOMCATV	original	377.6	-	-	-	-
	normal	405.2	3730204.0	16.00	2.90*4	7.31
	adaptive	405.6	3730716.0	16.40	2.90*4	7.42
	trained	386.0	3730204.0	0.40	2.00*4	2.22
NASA7	original	3662.0	-	-	-	-
	normal	3879.0	2924748.0	212.80	1.05*4	5.93
	adaptive	3916.4	2928744.0	250.20	1.05*4	6.95
	trained	3699.6	2924788.0	30.60	1.75*4	1.03

5. Implementation of the CATCH GCC Compiler

5.1. CATCH Filter and CATCH Libraries

The CATCH filter (Figure 1), placed between the parsing and optimization stages, works on an intermediate language representation called the Register Transfer Level (RTL). The CATCH filter executes the following:

1. Inserts the subroutine call *_prolog*() at the start of the *main*() function.
2. Inserts the subroutine call *_epilog*() at the end of the *main*() function.
3. Locates loops and subroutines to insert sparse potential checkpoint code as outlined in Section 2 and Section 4 when training is performed.
4. Replaces traditional library calls to *fopen*(), *fclose*(), *exit*(), and *abort*() with *_open*(), *_close*(), *_xit*(), and *_bort*() respectively.
5. Inserts variables that are used by the checkpointing scheme.
6. Controls the training mode (Section 4).

The CATCH library contains the corresponding definitions for the subroutines *_checkpoint*(), *_prolog*(), *_epilog*(), *_potential*(), *_snapshot*(), *_open*(), *_close*(), *_xit*(), and *_bort*(). In addition, an altered version of *malloc*() and the data compression/decompression code is included. The implementation details of the subroutines are described in the following subsections.

5.2. Checkpointing

The *_checkpoint*() subroutine saves the register environment and records the current position of each open file. A *reentry address* located towards the end of the *_checkpoint*() subroutine is then stored. The reentry address is the location were execution will resume at recovery time. The start and end address of the *dataseg*, which contains both the initialized data segment and the uninitialized bss segment, and the *spseg*, which is just the stack segment, are both determined. Finally, the layout of memory usage (i.e., the start and end

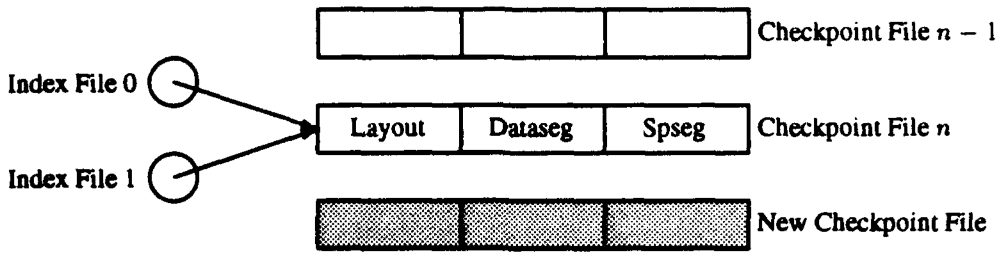

Figure 6: Checkpoint files and index files

addresses of dataseg and spseg) is saved in a *checkpoint file* along with the actual contents of dataseg and spseg (Figure 6).

A difficult issue in checkpointing is how to handle the state space associated with I/O states and disk contents. In practice, many user-level checkpointing methods will either not permit I/O operations [19] or only operate with a specially designed I/O subsystem [18]. Fortunately, for a majority of non-interactive applications, simple stream file I/O is only needed. With our CATCH implementation, sequential reads and writes of *stdin, stdout, stderr* or disk files using the *stdio* package (furnished in a UNIX system) are allowed.

In order to tolerate system crashes, a minimum of two checkpoint files (one working file plus one or more established files) need to be maintained. Each checkpoint file has a unique name, *base*.ckp$n$, where *base* is the base name of the file containing the *main*() function and n is the *sequence number* of the checkpoint file. A checkpoint file is considered *valid* only if all information has been recorded in the file and it has been successfully closed.

The most recent checkpoint file is kept track of by an *index*. A *setckp* utility is provided that allows the user or an automatic restart mechanism to modify the index for cases involving latent error detection. The index is recorded in two identical *index files* (Figure 6), named *base*.act0 and *base*.act1. Two indexes are maintained so that a system crash during the update of the index would not destroy a record of the index (see the following subsection). The steps involved in creating a new checkpoint are as follows:

1. Create the nth valid checkpoint file F_n.

2. Update the index value in the first index file I_0 from $n - 1$ to n.
3. Update the index value in the second index file I_1 from $n - 1$ to n.

The checkpoint is considered *established* when the content of I_0 is set to n in step 2.

5.3. Recovery

For a process to recover from an error, the process is simply restarted by reissuing the original command. The responsibility of reissuing can be handled by the user or by an automatic restart mechanism that monitors all submitted but unfinished jobs. Upon reissuing the original command, the process will immediately enter the *_prolog*() subroutine. Subsequently, the index value stored in I_0 (denoted i_0) and the index value stored in I_1 (denoted i_1) are read to determine which checkpoint file to use. Index file I_0 (or I_1) is *valid* if it exists as well as the checkpoint file that is points to, F_{i_0} (or F_{i_1}), also exists. The actions taken during recovery are guided by the list of rules:

1. If neither I_0 nor I_1 is valid, start execution from the beginning.
2. If I_0 is valid but I_1 is invalid, use F_{i_0} for recovery.
3. If I_0 is invalid but I_1 is valid, use F_{i_1} for recovery.
4. If both I_0 and I_1 are valid and $i_0 = i_1$, use F_{i_0} for recovery.
5. If both I_0 and I_1 are valid but $i_0 > i_1$, use F_{i_0} for recovery.
6. If both I_0 and I_1 are valid but $i_0 < i_1$, use F_{i_1} for recovery.

Before the end of the *main*() function, the *_epilog*() subroutine is encountered which deletes both index files and any remaining checkpoint files. All *abort*() and *exit*() subroutine calls are replaced by *_bort*() and *_xit*() respectively, which will call *_epilog*() before terminating the program. The *_open*() subroutine will call *fopen*() after recording the file name and access mode of the file. The *_close*() subroutine replaces the traditional *fclose*() call whose purpose is to delete the information recorded by

Table VI: CATCH compiler switches

switch	default	description
-catch-n	off $n = 1$	enable the CATCH filter, maintain at least n checkpoint files during normal execution
-auto-n	off $n = 60$	enable automatic insertion of sparse potential checkpoint code with $T = n$ seconds
-pl-n	on $n = 1000$	set reduction factor $l = n$
-show	off	display the checkpointing and recovery activities
-compress	off	enable LZW data compressor
-adaptive-n	off $n = 0$	set adaptive checkpointing window size $w = n$
-train-n	off $n = \infty$	set training mode set maximum training time to n

_open(). A modified version of *malloc*() exists that returns to the operating system the deallocated memory blocks contiguous to the unallocated area through *brk*() or *sbrk*().

5.4. User Control

The CATCH GCC compiler does permit the user to explicitly add checkpointing code to the program, if desired. A potential checkpoint location can be created by simply adding the subroutine *_potential*() to the code. The compiler will not add any sparse potential checkpoint code in a block where a *_potential*() subroutine is explicitly added. Similarly, a checkpoint location can be created by adding the subroutine *_checkpoint*() to the program. Conversely, if the speed of a particular block is critical to a program, the subroutine *_nocheckpoint*() may be added so that the compiler will not add any sparse potential checkpoint code in the block. Incidentally, no object code is generated for the *_nocheckpoint*() subroutine. The different checkpointing schemes with their corresponding parameters can be selected by the user by means of compiler switches. The switches, their description, and their default settings are shown in Table VI.

6. Conclusion

A means of performing checkpointing with the aid of a compiler was detailed in this paper. The compiler-assisted approach is transparent to both the programmer and the hardware. Application level checkpointing inserted sparse potential checkpoint code into the user program. For programs exhibiting large variations in memory usage, the adaptive checkpointing technique reduced checkpoint size as well as checkpoint time by allowing a window of opportunity to take a checkpoint. The training method selected low-cost, high-coverage potential checkpoints which had the effect of reducing the overall polling time overhead. Experimental results revealed that the CATCH compiler works well on code consisting of loops with many iterations and large loop body size.

References

[1] N. J. Alewine, S.-K. Chen, C.-C. Li, W. K. Fuchs, and W.-M. Hwu, 'Branch recovery with compiler-assisted multiple instruction retry', *Digest of Papers, FTCS-22,* Montreal, 1992, pp. 66–73.

[2] J. Long, W. K. Fuchs, and J. A. Abraham, 'Compiler-assisted static checkpoint insertion', *Digest of Papers, FTCS-22,* Montreal, 1992, pp. 58–65.

[3] V. Balasubramanian and P. Banerjee, 'Compiler-assisted synthesis of algorithm-based checking in multiprocessors', *IEEE Trans. Computers,* **39**, (4), 436–446 (1990).

[4] K. D. Wilken and T. Kong, 'Efficient memory access checking', *Digest of Papers, FTCS-23,* Toulouse, 1993, pp. 566–575.

[5] K. M. Chandy and C. V. Ramamoorthy, 'Rollback and recovery strategies for computer programs', *IEEE Trans. Computers,* **C-21**, (6), 546–556 (1972).

[6] S. Toueg and Ö. Babaoğlu, 'On the optimum checkpoint selection problem', *SIAM J Computing,* **13**, (3), 630–649 (1984).

[7] S. J. Upadhyaya and K. K. Saluja, 'An experimental study to determine task size for rollback recovery systems', *IEEE Trans. Computers,* **37**, (7), 872–877 (1988).

[8] R. M. Stallman, *Using and Porting GNU CC.* Free Software Foundation, Inc., version 2.4.3, 1993.

[9] K. M. Chandy, J. C. Browne, C. W. Dissly, and W. R. Uhrig, 'Analytic models for rollback and recovery strategies in data base systems', *IEEE Trans. Soft. Eng.,* **SE-1**, (1), 100–110 (1975).

[10] E. Gelenbe, 'On the optimum checkpoint interval', *J ACM,* **26**, (2), 259–270 (1979).

[11] A. N. Tantawi and M. Ruschitzka, 'Performance analysis of checkpointing strategies', *ACM Trans. Computer Systems*, **2**, (2), 123–144 (1984).

[12] C. M. Krishna, K. G. Shin, and Y.-H. Lee, 'Optimization criteria for checkpoint placement', *Comm ACM*, **27**, (10), 1008-1012 (1984).

[13] I. Koren, Z. Koren, and S. Y. H. Su, 'Analysis of a class of recovery procedures', *IEEE Trans. Computers*, **C-35**, (8), 703–712 (1986).

[14] K. G. Shin, T.-H. Lin, and Y.-H. Lee, 'Optimal checkpointing of real-time tasks', *IEEE Trans. Computers*, **C-36**, (11), 1328–1341 (1987).

[15] P. L'Ecuyer and J. Malenfant, 'Computing optimal checkpointing strategies for rollback and recovery systems', *IEEE Trans. Computers*, **37**, (4), 491–496 (1988).

[16] R. Geist, R. Reynolds, and J. Westall, 'Selection of a checkpoint interval in a critical-task environment', *IEEE Trans. Reliability*, **37**, (4), 395–400 (1988).

[17] B. Dimitrov, Z. Khalil, N. Kolev, and P. Petrov, 'On the optimal total processing time using checkpoints', *IEEE Trans. Soft. Eng.*, **17**, (5), 436–442 (1991).

[18] D. J. Taylor and M. L. Wright, 'Backward error recovery in a UNIX environment', *Digest of Papers, FTCS-16*, Vienna, 1986, pp. 118–123.

[19] J. M. Smith and J. Ioannidis, 'Implementing remote *fork()* with checkpoint/restart', *Technical Committee on Operating Systems*, **3**, (1), 15–19 (1989).

[20] C. V. Ramamoorthy, K. H. Kim, and W. T. Chen, 'Optimal placement of software monitors aiding systematic testing', *IEEE Trans. Soft. Eng.*, **SE-1**, (4), 403–411 (1975).

[21] C. F. Gerald and P. O. Wheatley, *Applied Numerical Analysis*, Addison-Wesley, Reading, MA, 1984.

[22] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge, MA, 1988.

[23] SPEC Steering Committee, *SPEC README file*, May 1990.

[24] T. A. Welch, 'A technique for high-performance data compression', *IEEE Computer*, **17**, (6), 8–19 (1984).

[25] D. E. Knuth, *The Art of Computer Programming, Vol. 3, Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[26] P. P. Chang and W.-M. W. Hwu, 'Trace selection for compiling large C application programs to microcode', *Proc. 21st Annual Workshop on Microprogramming and Microarchitecture*, November 1988, pp. 21–29.

[27] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, 1979.